N O T I C E

THIS DOCUMENT HAS BEEN REPRODUCED FROM MICROFICHE. ALTHOUGH IT IS RECOGNIZED THAT CERTAIN PORTIONS ARE ILLEGIBLE, IT IS BEING RELEASED IN THE INTEREST OF MAKING AVAILABLE AS MUCH INFORMATION AS POSSIBLE

alternatives are and their differences and similarities.

The tangible components of a development environment are *tools*, software which can be used in performing the development activities delineated above. Tools may be general ones which support a variety of activities, but more usually they have a purpose falling within one of the activities distinguished in this discussion.

The intangible components are the *development procedures, principles and practices* which development practitioners are encouraged or required to use. There is an obvious, close interrelationship between these methodological components of the environment and the tool components - the procedures, practices and principles included in the development environment shape the function and nature of the tools and the tools, themselves, give concrete meaning to the methodological rules and guidelines. The tangible and intangible aspects of a development environment must, therefore, be defined concurrently - proscribing either in isolation runs the risk of defining an infeasible or ineffective environment.

The core of a development environment is the *set of languages* provided for describing the characteristics, properties and functioning of a system during its development. These languages provide the media for communication, the basis for defining assessment techniques, and the means by which the relationships among alternatives may be explicitly or implicitly defined. Further, the languages allow the precise definition of the development environment's tool and methodological components in terms of how descriptions in the languages are prepared, modified, related and analyzed.

A broad spectrum of development environments is possible, depending upon the number and variety of languages and the extent to which a single development methodology guides the definition of the environment. At one end of this spectrum are *toolboxes* which provide a relatively large number of languages, generally having no well-defined interrelationships, and which support a variety of development methodologies. The tools included in a toolbox are either general ones, useable upon descriptions in any of the languages and therefore ignorant of the specific details of any particular language, or specific ones strongly linked to one particular language and usable only with descriptions in that language. The tools are also generally independent so that they may be employed in a variety of combinations in support of the variety of methodologies for which the toolbox can be used.

At the other extreme in the spectrum are *development systems*. These "union shops" of the development environment world provide a single, obviously very general, language and provide support for only a single methodology - practitioners who do not know (or like) the language or who do not subscribe to the enforced methodology cannot obtain help from this type of environment.

It should be obvious that the ideal development environment lies somewhere between these two extremes. These *development support systems* should provide a coordinated set of languages having well-defined relationships and features which allow the easy and natural expression of the properties, characteristics and functions of interest. This implies that development support systems generally cannot be universal and, rather, are oriented toward the development of a specific class of systems. Further, the tools provided by a development support system should be interdependent, integrated around a collection of methodologies which are similar in nature but which accommodate a number of development styles. The "happy medium" to be struck is one which encourages (and rewards!) good practices but does not stifle individuality. The characteristics of these ideal development environments will be discussed further in the last two sections.

PROGRAM CONSTRUCTION TOOLBOXES

Current-day software system development environments are, for the most part, of the toolbox variety and tend to support implementation activities occurring during the development of relatively simple systems. These *program construction environments* provide tools to aid in the development of an executable description of the program and in the determination of the program's run-time behavior. They tend not to embody any particular development methodology.

In order to suggest guidelines for producing development support systems, it is instructive to look at the components of current-day program construction environments and take note of several trends evidenced by their history. The first of these topics is the subject of this section and the second is discussed in the next section.

Our assessment of the current state of advancement in the preparation of program construction environments is oriented toward the facilities which the average development practitioner could expect (or could demand!) to find as part of their computing facility. They are, therefore, those facilities which are well enough developed to have been transferred out of "research" status - other facilities which are still under development are not discussed. It should also be emphasized that it is not necessarily feasible to implement all of the facilities indicated here on all computing systems and that their implementation on resource-impoverished computing systems is currently a research topic.

The typical program construction environment makes available a plethora of programming languages, generally enough to satisfy any taste, style or proclivity. In addition to general-purpose, procedural languages, languages are usually provided for specific problem domains (e.g., Snobol), specific solution spaces (e.g., Lisp), or specific solution activities (e.g., RPG). As is true of toolboxes in general, the languages are independent and no attempt has been made to define relationships among different languages.

The vast majority of tools present in a program construction environment provide support for text preparation activities. The task of translating a human-oriented, problem-oriented description into executable code is well-understood and translators of significant sophistication, in terms of the language constructs they can successfully handle and the optimization they can perform, are commonplace. Preprocessors, especially those providing structured-programming dialects of (generally older) languages lacking a reasonable set of execution control constructs, are also commonplace. Finally, for those wishing to prepare a translator for a language of their own design, there are a variety of parser and lexical analyzer generator tools.

Other, language-independent, text preparation tools are also available in current-day program construction environments. Chief among these are text editing systems of which there is an extensive variety. The best appear to be ones which are character-oriented and CRT-based; but line-oriented editors seem preferable when only hard-copy terminals are available. Subroutine libraries also provide help in preparing the text of a

program.

Text retention tools are usually present as part of the general-purpose file system provided by the operating system. Protection and access control facilities are supportive of program construction; the latter, in particular, being valuable in the enforcement of development principles such as the principle of information hiding [2]. Retrieval facilities are typically file-oriented and, therefore, not very sophisticated with respect to composing a program out of units much smaller than a procedure. This failing can, however, be circumvented with facilities which allow the free composition of files by placing directives in one file indicating that all or part of another file is logically part of the file containing the directive.

A variety of tools are typically provided to aid consistency assessment. Simple tools of this type are generally built into the language translators and check for syntax errors. Semantic error detection tools are also frequently provided as part of translators. Some of these check for simple errors such as a mismatch between arguments in a procedure invocation and the corresponding parameters. Others perform more sophisticated checks but require the users to give additional, semantic information so that the check can be carried out. For example, a units checker tool [3] has been developed for Pascal - it requires the specification of units, e.g., miles/hour, for program variables and then analyzes arithmetic expressions to determine whether or not units cancel properly.

Tools for checking the consistency between the program's overall dynamic (i.e., *run-time*) behavior and the program developer's expectations are primarily of the validation variety. This class of tools includes debugging support systems, path analyzers, testbeds, test coverage analyzers, test data generators, etc. These tools are typically language-specific. Also falling in this class are machine simulators which allow the execution-style checkout of a program even though the machine on which it will eventually execute is not available.

Other tools for checking a program's dynamic behavior by static verification techniques are not yet prevalent, but are becoming more common. Most generally available are those, typified by DAVE [4,5], which check for properties which stem from semantic rules (such as definition of a variable's value prior to its first use), from rules of good practice (such as use of a variable before redefinition of its value), or from general requirements pertaining to an entire class of systems (such as absence of deadlock). More sophisticated tools, typified by [6] and [7], use symbolic execution techniques and allow the checking of properties defined by the program developer and specific to the program itself.

The remaining development activities of completeness assessment, structure exploration and binding exploration are generally not directly supported by any specific tools. Rather, there is a hodgepodge of tools which have been "collected" over the years that tend to allow developers to investigate a program's completeness and explore alternatives. Generally included in this collection are cross-reference facilities, linkage editors, time-histogram facilities, etc. Specialized job control languages also provide support in carrying out these activities. Finally, additional support is sometimes provided by homogenizing programming language interfaces so that multi-lingual programs can be constructed.

TRENDS IN THE EVOLUTION OF PROGRAM CONSTRUCTION TOOLBOXES

The history of the development of program construction toolboxes exhibits some trends which, when extrapolated, indicate what can be expected in the future. We do not propose that extending development in the directions indicated by these trends will be sufficient to achieve truly effective development support systems. But the trends are indicative of an already established momentum which provides a context for other evolutionary trends which we will propose, in a subsequent section, as additionally necessary.

One trend that is quite obvious is toward *interactive* environments which provide the means for much quicker information transfer between developers and the computer, and vice versa. Once computer system users make the change from thinking of programs as a deck of cards to viewing their programs as text streams, interactive environments greatly speed up the activities of program text preparation, entry and modification. In addition, such environments offer a basis for speeding up the other development activities of assessment or exploration. The overall benefit is that developers are freed, by the virtue of higher-quality "secretarial services," to devote a larger proportion of their time to more important, more intellectually challenging development tasks. Of course, this benefit comes at some cost, but this is generally not prohibitive and recognized as well worth it.

Another trend is *integration*. With the development of sophisticated operating systems to serve as vehicles for the delivery of tools to development practitioners, there has been a tendency to provide a common interface to the various facilities provided within and under the operating system. Another aspect of integration has been provision of facilities which allow the various tools to be conveniently used in the combinations required under a number of development approaches and styles.

Closely related is a trend toward *unification*. An example is the development of debugging facilities which are oriented toward high-level rather than assembly-level languages; for example, the development of the debugging subsystem for the Algol-W language [8]. It is important to note that this trend toward unification generally implies a narrowing of scope of attention to a smaller number of languages, found by use over a number of years to be valuable, natural and sufficient for the representation of programs.

Closely related to these two trends is one which can be characterized as *centralization*. The tendency has been to provide new tools as part of a translator, perhaps optionally invokable. The usual, overriding reason for this is that the tool needs the facilities provided by the parser component of the translator and the (wise) decision is made to embed the tool in the translator rather than duplicate the parser. It should be noted, however, that when new language constructs are involved, the tool is frequently provided as a stand-alone facility; for example, structured programming preprocessors are the norm. However, the desire to make the tool independent of existing translators sometimes adversely affects the new language constructs.

Another trend, critical to providing program construction environments of any reasonable quality and effectiveness, has been toward *program animation* to provide help for the activities of assessment and exploration. Because of the availability of the machine on which the program will run (or a simulation of the machine), there is a strong temptation to use execution to gain an understanding of a program's dynamic behavior.

ORIGINAL PAGE IS
OF POOR QUALITY

Further, this "easy way" is encouraged because of the trend to centralize tools into a single programming system - it is frequently impossible, for example, to have the syntax of a program checked without also executing the program or at least having an executable translation of the source text prepared. There is increasing recognition, however, that the inference of a program's dynamic properties by execution is generally neither cost-effective nor effective.

The final trend to be commented on here concerns the environment's languages rather than its tools and can be described as a trend toward *higher-quality representation*. One aspect is a move toward increasing the understandability of programs. Founded upon a recognition that allowing an arbitrary relationship between the physical structure of a program and its logical flow (i.e., the paths of execution through it) unnecessarily increases the complexity of a program, and fostered by the desire to automatically verify a program's correctness, there has been both a general recognition that a programming language must provide a rich set of simple, powerful and easily understood execution control constructs and an increasing tendency in the definition of new languages to provide such a set.

Another aspect of understandability enhancement has been a move toward a separation of concerns. One simple example, evidenced by the Gypsy language [9] among others, has been the provision of constructs to explicitly control the scope of variables rather than have the block structure of the program indicate its resource requirements and the scope of variables. A more sophisticated example is provided by the constructs present in the Ada language [10] which allow a subprogram's intent (i.e., its specification) to be stated separately and redundantly of its implementation.

A closely related aspect is the tendency to introduce, into a language's definition, rules of usage that are relatively easily checked and which foster the production of "correct" programs. The thought behind this tendency is that a program's function is more easily understood, and more easily checked for validity, if aspects of its operation (and sometimes its behavior) are stated redundantly in all of those places where they are of interest. Constructs for explicit control of variable scope are an example, as are those for specifying the types of variables. It should be noted that facilities for redundant specification are usually accompanied by rules, enforced by the language's translator, requiring that the redundant specification be made by the language's users. This reflects the fact that these facilities stem from a realization that their use typically leads to "more correct" programs and that their imposition can speed the development of programs which function correctly.

Another closely related aspect is the trend in language and translator design to provide facilities supporting modularity, typified by incremental compilation facilities. These facilities are usually coincident with procedure definition facilities which means they are not always sufficient. But they do tend to facilitate incremental program development.

INADEQUACIES OF PROGRAM CONSTRUCTION TOOLBOXES

The general trend in the evolution of program construction toolboxes has been to provide facilities for hastening the development of appropriately functioning programs. Part of this has been the preparation of creation activity facilities which free development practitioners from devoting an inordinate amount of their efforts to the preparation and maintenance of the text of their program descriptions. Another part has been the introduction into programming languages of constructs fostering the development of correct, understandable programs. The final part of this trend has been the development of facilities for the consistency assessment of programs.

An obvious inadequacy is the lack of facilities providing direct, high-quality aid for completeness assessment and exploration activities. The problems associated with these activities have been somewhat less important and attention has been justifiably directed toward the more important problems associated with creation and consistency assessment activities. But it is also true that these problems have been intellectually managable for the vast majority of programs developed, and it is only with respect to the development of large-scale, complex programs (i.e., *software systems*) that these problems become unmanagable and aid is required.

When attention is turned to software systems as opposed to programs, several other inadequacies become apparent. These inadequacies primarily stem from a failure to take into account the special nature of systems that make them quite different from ordinary programs. Systems are a conglomeration of interacting parts; sometimes this decomposition of a whole into parts is a natural phenomenon and sometimes it is artificially induced because of our inability to otherwise cope with the system's complexity. Thus, all activities concerned with developing a system must consider the system's parts *and* their interactions and it is facilities for direct consideration of interactions that are absent from current-day program construction environments.

More specifically, what is missing are facilities to perform modelling. What is needed is the ability to focus upon the interactions, either as a prelude to developing parts which support these interactions or in an attempt to assess the interactions of already developed parts. This requires the ability to abstract the functional, operational characteristics of the parts, i.e., the ability to prepare models of a part's interface and functionality.

Some modelling capabilities are inherent in a program construction environment. It is possible, for example, to develop a system *prototype*, a simplified version which does not exhibit all of the properties of the eventual system but can be used in order to gain an understanding of systems of the type being developed. Use of models of this type can be called evolutionary programming [11] and can provide a very effective development method as evidenced by the MTS operating system [12] which was developed as a succession of progressively more elaborate "models." But it is equally important to be able to prepare *horizontal* models which represent the entire system but only to a level of detail which is somewhat short of an executable version of the system. These types of models are of particular importance during the early stages of development, by whatever development method is being employed, when the task is to infer or specify overall system properties without the ability (or necessity) of executing the system to determine its properties.

This leads to another inadequacy of current program construction environments, being the strong orientation toward functionality properties. During software system development, particularly when there are no previously developed, similar systems to provide hints concerning the system's eventual properties, practitioners need the capability to obtain estimates of the system's economics and performance. While this can be accomplished by implementing the system and gathering statistics during its

operation, this would be a regression to an ancient practice, described by Graham [13] as building systems like the Wright brothers built airplanes - constructing them, pushing them off a cliff, watching them crash and starting all over agian.

Another inadequacy has been implicit in the discussion so far - the inadequate recognition of pre-implementation development stages. During these stages, it is critically important to be able to unambiguously record and rigorously assess the policies and strategies governing the system rather than the mechanisms and algorithms used in its implementation. The languages provided are inadequate for expressing these attributes of a system and the tools provided do not support investigation of the system's properties as derived from these attributes. Further, the languages and tools are not as helpful as necessary or desirable during the post-implementation stages of maintenance and modification, which are more akin to pre-implementation stages than they are to implementation itself.

The final inadequacy to be noted here is in essence a secondary effect of the strong orientation toward the implementation stage of development. With respect to software systems, the development of software is no longer a personal thing between one person and the computer - there is, instead, a development team (sometimes more appropriately misspelled "teem"), as well as project managers, customers, users, acceptance testers, documentors, user-guide writers, etc. Each of these persons has differing descriptional requirements and this severely complicates the problems of communication and highlights the fact that programming languages and programming language oriented tools are not sufficient for an effective development support system.

PRINCIPLES GUIDING FUTURE EVOLUTION

We do not propose "starting all over again" in order to prepare effective development support systems. First, evolution, as opposed to revolution, is too well recognized as an efficient and successful paradigm. Second, revolution is not warranted, at this point, since program construction environments are basically on the right track and their inadequacies stem primarily from consideration of too small a set of concerns. Third, we feel that the trends evident in the evolution of program construction environments are, with only two exceptions, entirely appropriate and conducive to the eventual emergence of development support systems.

Therefore, in this section, we present several principles which we feel have affected the evolution of development environments only secondarily or not at all, but which must be given strong emphasis in order that the evolutionary process yields effective development support systems and that the emergence of such systems, and their delivery to working practitioners, is both quick and timely.

Principle 1: Enhance the expressive power and richness of the languages underlying the development environment.

The need to describe characteristics in addition to functionality and operation and the need to communicate essential information to a variety of audiences means that languages are needed to directly support a multiplicity of views of the system being developed. The traditional dual views of data flow and control flow are a simple illustration of what is needed - description capabilites providing alternative views and having a formal (although not necessarily algorithmically analyzable) relationship. Perhaps the ideal would be to have a single representational technique, not necessarily directly used by developers, from which all other system descriptions are analytically derivable. Whether or not this is an ideal, and what the various description techniques should be, will require a good deal of investigation.

In addition to evolving a set of coordinated, formally relatable languages, it is important that none of the languages exhibit a rococo nature. For example, the language which was developed as the basis of the DREAM design support system [14] is somewhat of this nature. In trying to put into one language, in consistent forms, all of the descriptional capabilities we felt necessary, we possibly created the PL/I of design description languages. The lesson learned is that it is much better to define a multitude of languages, each having a well-defined purpose, than it is to define a single language having a multitude of purposes - this follows the obvious extension of the trend toward separation of concerns evidenced by recent developments in programming language design.

Further, in the development of individual languages we should strive for a reasonable balance between sufficiency and naturalness. Having a parsimonious set of constructs is desirable because of the attendant clarity of the language and the relative ease with which one may obtain a formal basis for the language. Naturalness is obviously important but tends to increase the language's "size." In order to realize the aim of having formal relationships among the languages, it is perhaps best to err on the side of sufficiency.

One last word of caution is that we must be careful not to create a Tower of Babel situation as we cannot afford to forestall and inhibit progress by a "profusion of tongues." This analogy indicates that we should be as concerned with the meta-languages we use to define the languages as we are with the languages themselves.

Principle 2: Develop more extensive animation facilites, in close coordination with the development of languages.

The most challenging of development activities is assessment, taxing our inference capabilities to their utmost, particularly in the case of concurrent systems. We must, therefore, increase the facilities, whether they be simulation-based or analytic in nature, available for aiding the inference of a system's properties while it is under development and the estimation of the properties it will eventually exhibit when its development is complete.

So that we do not suffer decidability and computational complexity problems, the animation tools should be of a *feedback* variety. By this, we mean that the tools should derive information for the developers concerning the system's dynamic properties, but should not attempt to completely certify that a system exhibiting these properties is appropriate, leaving that task for the developers to perform by interpreting the derived information. Animation tools of this sort have been developed ([4,5,15] for example) and experience with their use indicates that while quality is a serious problem, they provide immeasurable assistance by uncovering potentially erroneous situations which would have escaped detection under a less-rigorous inspection done without the tool's aid.

That the development of animation tools and of description languages must be coordinated is obvious since each affects the form and content of the other. An

ORIGINAL PAGE IS
OF POOR QUALITY

additional reason is that, for effective system development, the processes of *synthesis* and *analysis* must be tightly interleaved so that assessment is a continuous activity, integrated with the activities of creation and exploration - otherwise, we are back with the Wright brothers again.

Principle 3: Give concentrated attention to providing tools which directly support exploration activities.

On the surface, this principle means that support must be developed for preparing and exercising system models, both prototypes and horizontal abstraction models. Additionally, however, differential elaboration of these models must be facilitated since it is in this way that alternatives for achieving some system part can be assessed within the context of the rest of the system. This assessment-in-context is critically important to any meaningful exploration of alternatives.

To facilitate and enhance the exploration of alternatives, it is also necessary to provide facilities for the aggregation and structuring of information concerning the system being developed. Underlying Dijkstra's development of the guarded command construct [16] was the important observation that the creation of a program is *not* an orderly process but is more generally a relatively random porcess in which computational details emerge in an order which does not correspond to the order in which they are performed during program execution. This phenomenon is even more true of the pre-implementation development stages and thus it is important to provide an environment under which pieces of information can be recorded in the order of their generation and structured into a coherent base of information from which developers can extract "chunks" composed of interrelated pieces of information.

Even more sophisticated facilities are important for the support of exploration activities. First, the information retention facilities must be able to distinguish and keep track of versions of a system, preferably with only straightforward, natural directives from the developers as to the relationships among pieces of information concerning the system. Second, the facilities must allow the modification of the system description within any relatively arbitrary "slice" through the information base. Finally, the facilities should ideally monitor and guide the information agglomeration process, checking for inconsistencies and missing information - this capability obviously demands a significantly more extensive understanding of the development process than we currently possess.

Principle 4: The process of natural selection should be facilitated at every opportunity.

This principle is obvious and should not need to be stated, but there is a frequent tendency to forget this basic tenet of evolution. It is useful, therefore, to point out some concerns relating to this principle which should be kept in mind.

First, experimentation is an absolute necessity. This means, however, that much more is required than the conduct of experiments intended to investigate the efficacy and efficiency of various practices, principles and procedures - that is, various methodologies. It means that individual tools must be constructed with attention to providing mechanisms for monitoring their individual and collective use. It means that the operating system environments through which the tools are delivered to development practitioners must allow the monitoring of tool usage and the collection of statistics on tool utilization. Finally, it means that an understanding must be developed of how to conduct the experiments, what data to collect, how to reduce the data to meaningful derived measurements, and how to use the results to guide future evolution and experimentation.

To facilitate evolution, it is also necessary to have an organization underlying the development environment which is conducive to the reorganization, extension and modification of the tools present in the environment. Users should be able to employ the tools in whatever sequence and combination seem warranted and productive given the intents of the tools. The tools themselves should be robust enough to function, at least to the level of reporting an error, in (perhaps bizarre) contexts not originally envisioned. Users should also be able to modify the tools, particularizing them to specific tasks. (This, however, raises some serious support questions.) Finally, it should be possible to easily add new tools to the environment.

To achieve this modifiability of the tools and their use, we feel that it is not appropriate to continue the trend toward centralization. In fact, it would be best, at this point, to unbundle the tools typically present in a programming system. The availability of a stand-alone parser, for example, would facilitate the development of other stand-alone tools while reducing the effort needed to produce them. Interface problems arise, but we feel they are solvable and the benefit gained is well worth it.

The process of natural selection further requires a management environment which facilitates and encourages the use of tools. In addition to sometimes well-founded suspicions as to the efficacy and efficiency of individual tools, practitioners also possess a "momentum" in the use of well-known practices and procedures which inhibits their adoption of new tools. Management should foster the surmounting of this inertia by indicating that the use of tools, even those without clearly established "credentials," is both expected and respected. This requires that management be willing to give the necessary monetary support to using *and* learning to use the tool. Management must also give monetary support to the task of toolsmithing and establish this as a respectable and desirable activity.

Critical to establishing an environment in which natural selection can easily run its course is the development of criteria by which tools may be judged. These criteria are necessary in the design of experiments, required so that practitioners can make intelligent choices of which tools to use and when, and almost prerequisite to management willingness to provide the necessary support and encouragement. However, waiting until these criteria are fully developed would introduce a debilitating delay and it is necessary that some risks be taken and there be a willingness to develop and refine the criteria concurrent with the experimental use of tools.

Principle 5: Encourage, but do not mandate, the use of development practices, procedures and principles.

At this point in the evolution of development methodologies, not enough is known as to their value in particular development situations to warrant their strict imposition. Rather, their value must be explored simultaneously with the exploration of criteria for tool usage and of tools themselves. In addition, we feel that it will *never* be appropriate to mandate the usage of a single set of principles, practices and procedures for

development because there will always be a wide variety of viable styles and a wide variance in practitioner sophistication and level of experience.

It seems best, therefore, to head toward development environments which embody a development *philosophy* rather than a strict methodology, which provide tools supporting a variety of styles consonant with the development philosophy, and which (perhaps subtly, perhaps blatantly) reward working within the guidelines of the philosophy.

Principle 6: Human engineering considerations should be of primary concern.

An environment which fosters natural selection, encourages experimentation and invites practitioners to use tools in support of their development activities cannot exist without the tools being easy and convenient to use. There should be relatively homogeneous interfaces to the individual tools. The tools should be oriented toward users who are unsophisticated with respect to computer science in order to foster their involvement in the development process, particularly during the preimplementation stages. The tools should provide users with extensive control over its functions and facilities so that their usage may easily be tailored to the task being done and the users' need.

With respect to human engineering concerns, we feel that the trend toward interactive environments is not necessarily a good one to pursue. The activities of assessment and exploration seem to be primarily off-line activities and the development environment would, therefore, seem best supported by remote-job-entry facilities coupled with high-speed, hard-copy output facilities. The exception would seem to be the use of interactive graphics devices, such as in the Tell system [17], to provide quick display of different system characteristics.

CONCLUSION

The transfer of technological developments from research status to use by "real-world" practitioners seems to fairly consistently require a ten-year period of time [18]. In this paper, we have attempted to propose and justify some principles guiding the preparation of software development environments which are critically necessary to achieving, and hopefully speeding, this transfer rate. We first reviewed the state-of-the-art of development environments, noting their strong orientation toward the construction of relatively simple software systems. We then indicated some of the trends evidenced by the evolution of these program construction environments and some of their inadequacies. These observations formed a basis for suggesting several principles (some at slight variance with previously established trends) to guide future evolution and arguing their criticality to achieving truly effective and efficient development support environments.

Our observations have been primarily with respect to development practitioners - specifiers, designers, implementors, and maintainers. They have some validity with respect to other agents - managers, users, customers, acceptance testers, etc. - participating in the development process, but the special concerns of these agents have not been given adequate attention here.

Because of the quasi-research, quasi-development nature of the task of continuing the evolution of development environments, it would appear that the most success would result from university/industry or university/government collaborative efforts. Preliminary implementations and feasibility studies, tasks that industry and government software development divisions frequently cannot devote time to carrying out, could be performed in university environments guided by the practical experience of industry personnel. Production-version implementation and effectiveness assessment, tasks that university projects frequently lack the resources to adequately attack, could be performed in industry or government with the guidance of university researchers, particularly with regard to the conduct and interpretation of experiments. It would seem that the differing interests, experiences and capabilities of the various segments of the software engineering community are not extensive or rich enough for any one segment alone to meaningfully attack the overall problem; and it would seem that a collective effort would be effective and beneficial.

ACKNOWLEDGMENTS

Many of the observations made here emerged during the presentations and discussions at the Software Development Tools Workshop held in May, 1979, at Pingree Park, Colorado, and the author is indebted to the workshop participants for allowing him to "pick" their brains. The author would like to also thank Guy Bristow, Bryan Edwards, and Jack Wileden for helpful discussions.

REFERENCES

1. G. M. Weinberg. The Psychology of Computer Programming. Van Nostrand Reinhold Co., New York, 1971.
2. D. L. Parnas. Information distribution aspects of design methodology. Proc. IFIP Congress 71, Ljubljana, August 1971, pp. TA3/26 - TA3/30.
3. S. H. Saib. SQLAB: Tools for program verification. Proc. NASA Workshop on Tools for Embedded Computing Systems Software, Hampton, Virginia, November 1978, pp. 117-120.
4. L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. Computing Surveys, 8, 3 (1976), 305-330.
5. G. Bristow, C. Drey, B. Edwards and W. Riddle. Anomaly detection in concurrent programs. Proc. 4th International Conf. on Software Engineering, Munich, September 1979.
6. L. A. Clarke. A system to generate test data and symbolically execute programs. IEEE Trans. on Software Engineering, SE-2, 3 (September 1976), 215-222.
7. W. E. Howden. DISSECT: A symbolic evaluation and program testing system. IEEE Trans. on Software Engineering, SE-4, 1 (January 1978), 70-73.
8. E. Satterthwaite. Debugging tools for high level languages. Software - Practice and Experience, 2, 3 (July 1972), 197-217.
9. A. L. Ambler, D. I. Good, J. C. Browne, W. F. Burger, R. M. Cohen, C. G. Hoch and R. E. Wells. Gypsy: A language for specification and implementation of verifiable programs. Software Engineering Notes, 2, 2 (March 1977), 1-10.
10. Preliminary Ada Reference Manual. SIGPLAN Notices, 14, 6 (June 1979).
11. C. Hewitt. Remarks at Software Development Tools Workshop. In Riddle and Fairley (ed.), Software Development Tools, Springer-Verlag, Heidelberg, to appear February 1980.

ORIGINAL PAGE IS
OF POOR QUALITY

12. An Introduction to M.T.S. Computing Center, University of Michigan, Ann Arbor.

13. P. Naur and B. Randell (ed.) Software Engineering. Scientific Affairs Div., NATO, Brussels, Belgium, January 1969.

14. W. E. Riddle, J. C. Wileden, J. H. Sayler, A. R. Segal and A. M. Stavely. Behavior modelling during software design. IEEE Trans. on Software Engineering, SE-4, 4 (July 1978), 283-292.

15. P. Henderson. Finite state modelling in program development. SIGPLAN Notices, 10, 6 (June 1975), 221-227.

16. E. W. Dijkstra. General commands, nondeterminacy and the formal derivation of programs. Comm. ACM, 18, 8 (August 1975), 453-457.

17. P. G. Hebalkar and S. N. Zilles. TELL: A system for graphically representing software design. Proc. Compcon Conf., San Francisco, 1979.

18. C. A. R. Hoare. Keynote Address. Proc. 3rd International Conf. on Software Engineering, Atlanta, Georgia, May 1978.